KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculty of Business and Economics

A petri net formalization of a publish-subscribe process system

Pieter Hens, Monique Snoeck, Geert Poels and Manu De Backer

DEPARTMENT OF DECISION SCIENCES AND INFORMATION MANAGEMENT (KBI)

KBI 1114

A Petri Net Formalization of a Publish-Subscribe Process System

Pieter Hens
K.U.Leuven
Dept. of Decision Sciences and Information Management
Leuven, Belgium
pieter.hens@econ.kuleuven.be

Monique Snoeck
K.U.Leuven
Dept. of Decision Sciences and Information Management
Leuven, Belgium
monique.snoeck@econ.kuleuven.be

Geert Poels
Universiteit Gent
Dept. of Management Information and Operations Management
Gent, Belgium
geert.poels@ugent.be

Manu De Backer
Universiteit Antwerpen
Dept. of Management Information Systems
Antwerpen, Belgium
Manu.DeBacker@ua.ac.be

ABSTRACT

Publish/subscribe systems are getting more and more integrated into the execution of business processes in process aware information systems. This integration enables the distribution of the process logic and increases the scalability and adaptability of the process enactment infrastructure. A consequence is however that the original specified process model doesn't accurately represent the actual running process anymore, as the publish/subscribe specific operations are not incorporated into the original model. In this paper we propose a formal model of a publish/subscribe system that can be integrated into a business process model, creating in this way an accurate representation of the actual runtime process. The resulting model can be used for model checking the executable process: inspect system properties, discover problems and validate changes.

1. INTRODUCTION

Process-aware information systems (PAISs) are becoming increasingly integrated in todays business environments [8]. Companies are aware of the running processes in their organization, where they analyze, model and execute these processes. Together with Service Oriented Architectures, these processes can be executed automatically by a process engine. Executing a process logic means coordinating the described work, invoking the correct services, adding tasks to the inbox of task managers, and choosing the correct control flow paths [5].

This execution of process models is getting more and more incorporated with event driven architectures and publish/-subscribe communication paradigms, in order to leverage the advantages of event communication to process execution [27, 19, 11]. The decoupling features of event communication are used to increase the scalability and adaptability of the process execution architecture and are used to distribute business activities throughout the IT infrastructure (see Section 1.2 for an example) [11]. A consequence, however, of changing the process execution and adding publish/-subscribe communication inside the process is that the original process model doesn't represent the actual running process execution anymore. The dynamics and non-determinism of the publish/subscribe architecture itself is not represented in the original model. This influences *model checking* the implemented business process. With *model checking*, developers can discover deadlocks, livelocks, race hazards, ... or figure out system properties prior to actual process execution [4]. Model checking should be done on a model that stands closest to the actual implementation, a model that simulates the real process execution at its best. In this case, this means including pub/sub specific operations: *publish*, *subscribe*, *notify*, ... into the process model. This way the states of the publish/subscribe architecture are also included in the process model so that model checking can be performed on a more *complete* and *accurate* model.

To enable model checking on the combination of the original process model and the publish/subscribe communication architecture, a formal model has to be available that (1) simulates the behavior of the publish/subscribe system; and (2) is compatible with the original developed business process model, so that two models can be combined into one. In this paper we propose such a formal model expressed in a petri net. A petri net representation is used because the semantics of petri nets are clearly and formally defined and many analysis techniques exist for petri net model checking [21]. A second advantage of using petri nets as formal publish/subscribe model, is that in the business process modeling domain, petri nets are widely adopted as formal

notation to depict business process models [28, 25]. Because the same language is used to represent the business process model as well as the publish/subscribe runtime architecture, integration of the publish/subscribe formalism into the business process model can easily be achieved.

The publish/subscribe petri net model allows us to:

1. formally model a publish/subscribe connection between two or more communicating components in a business process;
2. *model check* a process execution which uses a publish/-subscribe communication paradigm between different components in the business process (check correctness criteria, system properties, ... for example, "Is the system still working correctly when event notifications are significantly delayed?"); and
3. verify the correspondence (correctness) of the actual running publish/subscribe process with the original specified business process model.

The following section briefly describes related work on model checking publish subscribe systems, followed by an example of the integration of a publish/subscribe architecture into process execution. Section 2 gives background information about publish/subscribe systems and petri nets. In this section, properties are defined which should be fulfilled by the petri net publish/subscribe representation. Section 3 then explains the petri net publish/subscribe model and checks if the event system properties defined earlier are satisfied by the petri net model (Section 3.1). Section 4 ends the paper with an explanation and some examples on the applicability (model checking) of the petri net publish/subscribe representation.

1.1 Related Work

Model checking publish/subscribe systems has been described by previous researchers. For example, Deng et al. [7] provides a model checking method for a CORBA event service in real time embedded software, whereas Garlan et al. [10] provide a more generic publish/subscribe modeling framework. Both are however not compatible with any process modeling languages used in business process modeling. Li et al. [17] provide a π-calculus formalization of publish/-subscribe systems in a Service Oriented Architecture. π-calculus is compatible with some process languages (e.g. BPEL [23]), whereas others and more current business process languages are based on petri net token semantics (e.g. BPMN2.0 [24] and YAWL [29]). The latter are by consequence better integrable with a petri net publish/subscribe formalization. Zanolin et al. [31] provide a modeling method for publish/subscribe systems using UML statecharts, in order to model and understand the global working of the system. The difference with our research is that we already start with the knowledge of the working of the global system (the process model) and need a way to model the inner workings of the publish/subscribe architecture itself.

Petri nets are also widely used to represent communication mechanisms and protocols between different components.

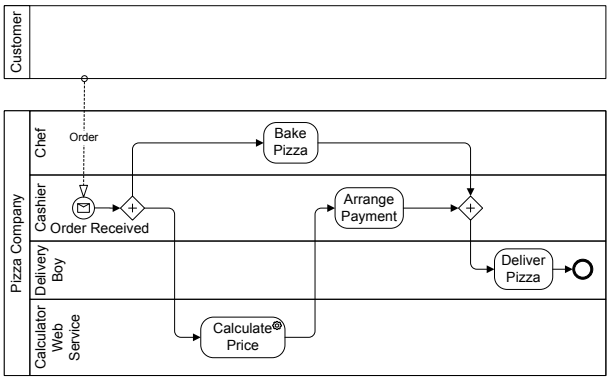


Figure 1: Business Process of a Pizza Company

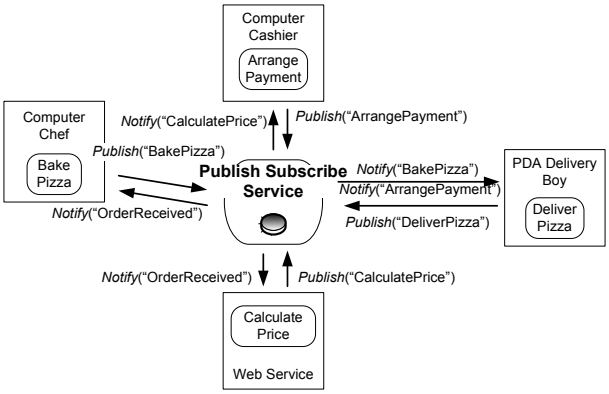


Figure 2: Decentralization with a publish/subscribe architecture of the process flow of the Pizza Company

Petri nets are used to model and check e-commerce protocols [15], C3I communication systems [18], polling systems [13], ... We are similarly modeling such a communication petri net, but this time for a publish/subscribe process execution.

1.2 Running Example

Figure 1 shows a business process for a pizza delivery company depicted in the Business Process Modeling Notation [24]. The process consists of four activities: one automated activity (*Calculate Price*) and three manual activities. An automated activity is handled by a (web) service, which can be invoked with technologies like SOAP and WSDL [6]. The manual tasks are handled by one or more task managers. If a manual activity needs to be executed, a task is added to the inbox of a specific actor. The actor can indicate the completion of the task, in which case the next activity in the process flow will be executed.

The model describes that the *bake pizza* activity can happen in parallel with the *calculate price* and *arrange payment* activities, while the deliver pizza activity has to wait before both preceding activities are complete. Every activity has been assigned a specific actor that can execute the business activity.

The distributed execution of this process flow is shown in

Figure 2. Each business activity is run on a dedicated process engine [3, 22, 11] (the cashier's computer for *arrange payment*, a web service for *calculate price*, the delivery boy's PDA for *deliver pizza*, ...) and a publish/subscribe architecture is used to accomplish the communication between the different components. This distributes the process logic throughout the IT infrastructure and increases the scalability of the process enactment architecture.

Because of the discrepancy between the original described business process model (which only defines an ordering relation between the different business activities) and the actual running process (which includes the publish/subscribe operations), a model is needed to represent the latter (Figure 2).

2. BACKGROUND

Before we demonstrate the publish/subscribe connection in a petri net, we introduce the properties of a publish/subscribe system and describe the semantics of a petri net. These are used to formally define the publish/subscribe representation and validate its correctness.

2.1 Publish/Subscribe system

In a publish/subscribe system, each communicating component subscribes to specific events it likes to receive notifications from, with an event being *the occurrence of a happening in the information system*. Such an occurrence can be the completion of a task, a database call, an alert, ... Management of the event-subscriptions is done by an (distributed) event system.

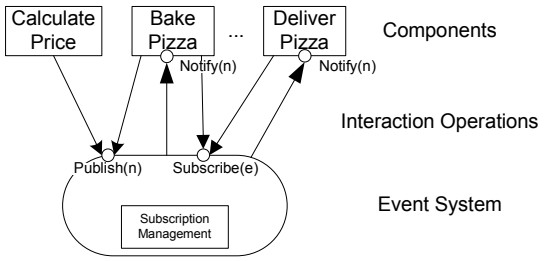


Figure 3: Basic Event System

Figure 3 shows a basic event notification service with publish/-subscribe operations from interacting components. Each component in the event-based architecture can *subscribe* to specific event notifications *published* by other components (*many-to-many communication*). In this case, we restrict one subscription to one event (*content-based subscription*, e.g. subscribe for the message with content 'end of activity *Bake Pizza*')[1]. After subscription, the component will get notified of any published notifications indicating the occurrence of that specific event.

[1]In many publish/subscribe systems, subscriptions are handled *topic-based*. In these systems, published notifications matching a topic (or channel) will be delivered to the subscribed component. Any (known or unknown at subscription-time-) message that matches the topic will be delivered to the subscribed component. Such a global subscription mechanism isn't necessary for our purposes. A mechanism where one subscription matches one (known at subscription time-) event is sufficient.

Table 1: Interface operations of a basic event system

Operation	Description
Publish(X,n)	Component X published notification n : $P'_X = P_X \cup \{n\}$
Subscribe(X,e)	Component X subscribes for all notifications of event e : $S'_X = S_X \cup \{e\}$
Unsubscribe(X,e)	Component X unsubscribes for all notifications of event e : $S'_X = S_X \backslash \{e\}$
Notify(X,n)	Component X is notified about notification n

Table 2: Temporal Logic Operators

$\Box$	The statement succeeding the operator is *always true*
$\Diamond$	The statement succeeding the operator will *eventually be true*
$\bigcirc$	The statement succeeding the operator will be *true in the next step*

2.1.1 Event System properties

We define the properties of an event system so as to know what should be satisfied by the petri net event representation. The following properties are based on the formal representations of a publish/subscribe system given in [20] and [1].

Definition 1. A publish/subscribe system is a tuple $<\mathcal{C}, \cup_{X\in\mathcal{C}} S_X, \cup_{X\in\mathcal{C}} P_X, \mathcal{N}, \mathcal{E}, E, Operations>$, with $\mathcal{C}$ the set of all communicating components, $\mathcal{E}$ the set of all events, $\mathcal{N}$ the set of all notifications for a specific event, $E : \mathcal{N} \rightarrow \mathcal{E}$ a unary function that maps a notification to the event the notification represents, S_X the set of active subscriptions for component $X \in \mathcal{C}$, with $\forall s \in S_X : s \in \mathcal{E}$ (one subscription relates to one specific event), P_X the set of all published notifications by component $X \in \mathcal{C}$, with $\forall n \in P_X : n \in \mathcal{N}$ and *Operations* the set of supported interface operations.

Four basic interface operations that constitute a simple event system are defined in Table 1 [20]: notifications can be published, components can subscribe and unsubscribe for consummation of specific events and notifications can be notified to components. The interface operations are however not limited to these functionalities, some event systems add for example an `advertise` operation (which *subscribes* publishers). In this paper we only work with the four basic operations defined in Table 2, but extra functionality can easily be added to the model.

Below, the safety and liveness conditions are given for a publish/subscribe system [20]. Safety and liveness conditions are properties from a concurrent system, where the safety condition states that nothing *bad* should happen and the *liveness* condition states that eventually, something *good* will happen (see Table 2 for a definition of the temporal logic operations used).

Definition 2. A publish/subscribe event system satisfies

the following requirements:

Safety

$$\Box\Big[notify(Y,n) \Rightarrow \big[E(n) \in S_Y\big] \wedge \big[n \in \cup_{X \in \mathcal{C}} P_X\big] \wedge \big[\bigcirc \Box \neg notify(Y,n)\big]\Big] \quad (1)$$

Liveness

$$\Box\Big[\Box(e \in S_Y) \Rightarrow \big[\Diamond \Box \big(publish(X,n) \wedge E(n) \equiv e \Rightarrow \Diamond notify(Y,n)\big)\big]\Big] \quad (2)$$

The safety condition declares that a notification should never be delivered to a component when (1) this component doesn't have an active subscription for that notification; (2) the notification isn't published first; and (3) the notification isn't already delivered to the same component (a notification can't be delivered more than once). On the other hand, the liveness condition declares that when a component is subscribed for an event (and never unsubscribes) and a notification for the subscribed event is published, that notification will be delivered to the component. In other words, the liveness condition requires the delivery of a notification, under the condition that a component never unsubscribes. The 'never unsubscribes' condition is added to take processing delays of the event system into account.

Besides the interface operations and the safety and liveness conditions, the decoupling features of an event architecture should also be supported. Decoupling of an event system is defined by Eugster [9] as space decoupling (unawareness of interaction partners), time decoupling (interaction partners don't need to be active at the same time) and synchronization decoupling (asynchronous send and receive). A consequence of this decoupling however (and more specifically time decoupling), is that it adds another form of non-determinism to the communication system. A concurrent communication system is inherently non-deterministic because of failures, message transfer delays, concurrent execution of processes, the order of message receipt, ... On top of these, time decoupling adds another cause for non-determinism: an event notification can be published, while a subscriber for this event is offline, or the subscriber can choose to consume the notification at a later time. In both cases, the notification can get lost when the event system doesn't store published, but not consumed notifications. A mechanism to store these notifications needs to be incorporated in the event system, in order to support the time decoupling flexibility.

Storage of event notifications has a twofold interpretation, do we need to store every notification ever published so that they are still available for any, eventual, subscriber? Or is it only necessary to store notifications for a specific component, that are published during the subscription time of that component (so they can be consumed at a later time)? The latter interpretation is beneficial when there are for example memory limitations (not every notification ever published should be stored). However, the first interpretation adds the ability to receive *historic notifications* (before subscription time). Both interpretations of storage (time decoupling) are supported by our petri net representation (see section 3).

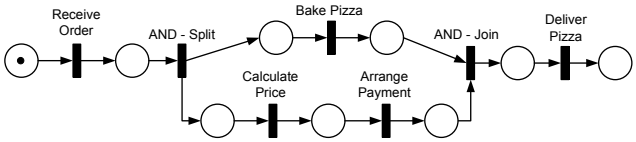


Figure 4: A Pizza Delivery process in Petri Net

2.1.2 Summary

1. The event system should support many-to-many communication;
2. The event system should support all the interface operations described in Table 1;
3. The safety- (equation 1) and liveness- (equation 2) conditions should hold at all times; and
4. Time, space and synchronization decoupling should be realized by the event system.

2.2 Petri Nets

Definition 3. A Petri Net is a tuple $PN = <P, T, F, \mu_0>$ with, P a finite set of places,
T a finite set of transitions, with $P \cap T = \emptyset$,
$F \subseteq (P \times T) \cup (T \times P)$ is the flow relation,
$\mu_0 : P \to \mathbb{N}$ is the initial marking of the petri net.

Places are graphically represented as circles, transitions as boxes and the flow relation as a directed arc between places and transitions (see for example Figure 4). The preset of an element $a \in (P \cup T)$ is defined as $\bullet a = \{a' | (a', a) \in F\}$ and the set $a\bullet = \{a' | (a, a') \in F\}$ is its postset. A marking of the petri net is a function $\mu : P \to \mathbb{N}$, which maps a place to a non-negative number. If $\mu(p_i) > 0$, place p_i is said to hold one or more *tokens*. The dynamic behavior of a petri net is defined as follows: (1) a transition $t \in T$ is enabled in a marking μ iff $\forall p \in \bullet t : \mu(p) \geq |(p,t) \in F|$; and (2) an enabled transition t can fire (execute), such that a new marking μ' is achieved according to the following rule (firing rule): $\forall p \in P : \mu'(p) = \mu(p) - |(p,t) \in F| + |(t,p) \in F|$.

Besides this classical definition of Petri nets, we use the extension that adds a color or data value to a token (Colored Petri Nets [14]). This higher level petri net allows us to add conditional expressions on flow arcs, as well as to define data concepts that are incorporated in tokens (or event notifications).

In process modeling these colored petri nets are used to model, execute and analyze workflows and workflow systems [28]. Figure 4 shows the pizza companies process depicted in a petri net. The process contains the four business activities and two control nodes (AND-split and AND-join), all represented by transitions in the petri net.

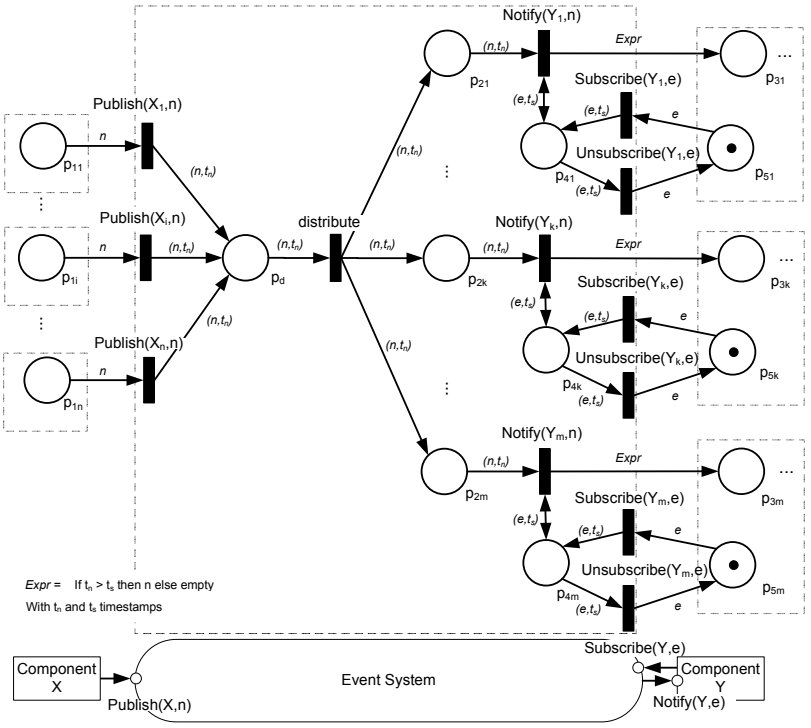


Figure 5: A publish/subscribe connection for event e, with $E(n) \equiv e$

3. PUB/SUB SYSTEM IN PETRI NET

Figure 5 shows the petri net representing event communication for one specific event e. The petri net represents a *publish-subscribe connection* between publishers of an event e (left hand side of the figure, places $p_{11}..p_{1i}..p_{1n}$) and subscribers of that specific event (right hand side of the figure, places $p_{31}..p_{3k}..p_{3m}$). The places and transitions inside the dashed square in the middle of the publish/subscribe connection, belong to the publish/subscribe connection itself. A publish-subscribe connection is made for each specific event $e \in \mathcal{E}$. The entire pub/sub system thus consists of all the *publish/subscribe connections* for each event in $\mathcal{E}$, this is noted as the set $\mathcal{PSC}$.

Each connection $c \in \mathcal{PSC}$ provides the interface operations necessary to communicate a notification for a specific event from the publisher to the subscriber. These interface operations (`publish(X,n)`, `subscribe(Y,e)`, `unsubscribe(Y,e)` and `notify(Y,n)`) are represented by transitions in the petri net event system. One extra transition `distribute` is added to the internal working of the event system, which transfers the published notifications to a buffer for every component subscribed to the event (places $p_{21}..p_{2k}..p_{2m}$). This buffer is used to enforce time decoupling. When a subscribed component is not online, the published notification is still available in the buffer place of that subscribed component (represented by a token). When the component comes back online, he can consume all the tokens available in the buffer place by executing the `notify(Y,n)` transition (no event gets lost)[2].

The condition on the flow arc leaving the `notify(Y,n)` transition ensures that only events which are published during the time period of subscription are delivered to the subscribed component. In the case that it is also necessary to receive *historic events*, which are published before the subscription time of a component (see Section 2.1.1), the condition on the flow arc leaving the `notify(Y,n)` transition can simply be removed.

The publish/subscribe connection assumes that the components, which communicate with each other through events, are known at design time. These components are modeled as publishers and subscribers in the publish/subscribe connection (left and right hand side of Figure 5). However, one of the advantages of using event communication is that run time changes can be made to the connections. More subscribers can be added at run time, or more publishers of a specific event notification can be added in an ad-hoc way. We allow these dynamic changes to the publish-subscribe connection, by permitting the following runtime changes to the petri net event system:

[2]Note that being online or offline is not represented as a transition (operation) in the petri net model. Due to unaccepted failures or crashes, it is not possible to assume that a component will always indicate his online or offline status through an operation call to the event system. We simply assume that the `notify` transition is only triggered by an active component (and never for offline components), e.g. if a timeout is reached when trying to notify a component, the event system will keep the token in the buffer place, and try again at a later time.

Definition 4. Dynamic changes in a petri net pub/sub system.
Let

$EC : \mathcal{PSC} \rightarrow \mathcal{E}$ A unary function that maps a publish-subscribe connection to the event for which this publish-subscribe connection exists.

Add publisher A new publisher X of notifications representing event e, can be added to the publish-subscribe connection $\{c|c \in \mathcal{PSC} \wedge EC(c) = e\}$ by creating a new transition `publish(X,n)`, with $\bullet$`publish` $= p_i \in X$ and $publish\bullet = p_d \in c$.
If a publish-subscribe connection for event e does not yet exist: $|\{c|c \in \mathcal{PSC} \wedge EC(c) = e\}| = 0$ a new connection c, with EC(c) = e has to be created first.

Remove Publisher Remove the transition `publish(X,n)` in the event system, together with its preceding and succeeding flow relations.

Add subscriber A new subscriber Y for an event e can be added to the publish-subscribe connection $\{c|c \in \mathcal{PSC} \wedge EC(c) = e\}$ by creating the interface transitions `notify(Y,n)`, `subscribe(Y,e)` and `unsubscribe(Y,e)` and new places $p_{2q} \in c$, $p_{4q} \in c$ and $p_{5q} \in Y$, with `distribute`$'\bullet$ = `distribute` $\bullet \cup p_{2q}$, $\bullet$`notify` $= p_{2q}$, , `notify`$\bullet = p_i \in Y$, `subscribe`$\bullet$ = p_{4q}, $\bullet$`unsubscribe` $= p_{4q}$, $\bullet$`subscribe` $= p_{5q}$, `unsubscribe`$\bullet = p_{5q}$ with $\mu(p_{5q}) = 1$.
If a publish-subscribe connection for event e does not yet exist: $|\{c|c \in \mathcal{PSC} \wedge EC(c) = e\}| = 0$, a new connection c, with EC(c) = e has to be created first.

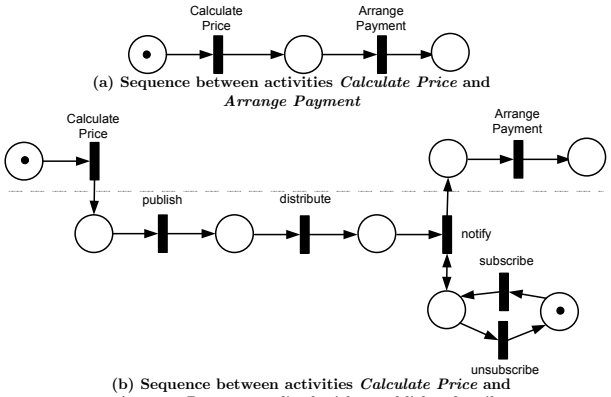
Remove subscriber. Remove the interface transitions `notify(Y,n)`, `subscribe(Y,e)` and `unsubscribe(Y,e)`, together with its preceding and succeeding flow relations.

The model defined in Figure 5 can now be used to define a model which includes the publish/subscribe operations, and simulates the real running process execution. Figure 6b shows an example of the publish/subscribe connection between the two succeeding activities *Calculate Price* and *Arrange Payment*. The component *Calculate Price* is connected with the `publish` transition of the publish/subscribe connection, while the component *Arrange Payment* is connected with the `notify` transition of the publish/subscribe connection. In the same way, the publish/subscribe connection can be applied to every connection between two components in the pizza companies process[3].

3.1 Checking the Pub/Sub Properties

We need to check that all the properties of an event system listed in Section 2.1.1 are fulfilled by the petri net representation of the pub/sub system. Remember that the entire publish/subscribe system is represented by the set of all publish/subscribe connections $\mathcal{PSC}$.

[3]Note that we have developed an algorithm that performs this translation automatically [12].



(a) Sequence between activities *Calculate Price* and *Arrange Payment*

(b) Sequence between activities *Calculate Price* and *Arrange Payment* realized with a publish-subscribe connection

Figure 6: Sequence between two activities *Calculate Price* and *Arrange Payment*, without and with the use of a publish-subscribe connection

Many-to-many communication. Tokens can be fired from multiple publishers with the use of their own publish transition for one publish/subscribe connection $c \in \mathcal{PSC}$. These fired tokens are routed to every place p_{2k}, where they can be used to fire every `notify` transition in the publish/subscribe connection. Each notify operation is inherently connected to a distinct component Y_k. Many publishers are thus allowed in the connection, as well as multiple subscribers.
With the dynamic changes defined, multiple publishers or subscribes can also be added at runtime to the connection.

Supporting the interface operations. Every interface operation listed in table 1 is represented as a transition in the publish-subscribe connection. The set of publishers for a specific event e in a connection $c \in \mathcal{PSC}$, is represented by the set of places in c : $\{p_{1i}\}_{i=1..n}$ (see Figure 5). The set of subscribers for a specific event is represented as the places $\{p_{4i}|\mu(p_{4i}) > 0\}_{i=1..m}$.
Besides explicitly modeling the subscribe and unsubscribe operations as transitions, it is also possible to leave these out and interpret a subscribe operation as the dynamic change of adding a subscriber (see definition 4). An unsubscribe operation is then the dynamic change of removing the subscriber. This way, only active subscribers are visible in the publish-subscribe connection, and no explicit subscribe and unsubscribe transitions are necessary.

Safety. The notify transition is only enabled in marking μ where $\mu(p_{4k}) \geq 1$ and $\mu(p_{2k}) \geq 1$. Because of the firing rule, marking $\mu(p_{4k})$ is only achieved when $\exists \mu'(p_{5k}) : \mu'(p_{5k}) \geq 1$, which enables the subscribe transition. This means that in order to achieve the firing of the `notify` transition, there should have been a firing of the `subscribe` transition, and not of the `unsubscribe` transition. This complies with the first part of the safety condition (a consumer of an event notification should be subscribed to that event).
The second part, the notification should have been published first, can be seen in the same way. In order to achieve the marking $\mu(p_{2k})$, transitions `distribute`

and `publish` should have happened first. This can also be seen from the state transition graph from a publish-subscribe connection. This graph is shown in figure 8. You can see that a publish and a subscribe transition always happen before a state is reached where the notify transition can occur.
The last element of the safety condition states that a notification can't be consumed, or notified twice to the same subscriber. This is also enforced by the firing rules. A token in the buffer place p_{2k} represents one notification published and distributed to the consumer. When the notify transition occurs, the token is fired and removed from the buffer place p_{2k} (see Section 2.2). The same notification (or token) is therefor never consumed (`notify` transition) more then once by the same component.

Liveness. The liveness condition is also enforced with our petri-net representation. An easy way to check this is with the state transition graph (Figure 8). If a publish transition and a subscribe transition happens, the graph always converges to a state where the notify transition occurs.

Decoupling. A petri net inherently represents a concurrent system, and is thus also non-deterministic. Multiple transitions can be enabled at any time and any of these transitions can be chosen to fire. Multiple tokens from multiple publishers can be present in the buffer place p_{2k}, and there is no order on which token is consumed first (a later published notification can be consumed before an earlier published notification). Time decoupling is achieved by the buffer place p_{2k}, which makes sure that no events get lost. Space decoupling is achieved because an intermediate system takes care of the communication between two interacting partners. A component publishes his notification (token) to the publish/subscribe connection and not directly to the subscribing component. Synchronization decoupling is also achieved, because tokens are asynchronously send from p_{1i} to p_{3l}. After the `publish` transition, the publisher-component continues its work and does not wait until the token arrives at the subscriber.

4. APPLICABILITY

The above defined petri net model of a publish/subscribe connection can now be used to check the correctness of changing a process model to a distributed system, where two or more activities (or components) from the original process model communicate in a publish/subscribe fashion.

The publish/subscribe connection for process execution opens the possibility to check two things:

1. It can be checked if the behavior of the resulting publish/-subscribe communication scheme is equivalent with the behavior of the original process model (as modeled by the process modeler). This will validate the use of a publish-subscribe communication system in process execution. An example of this equivalence checking is shown in Section 4.1.

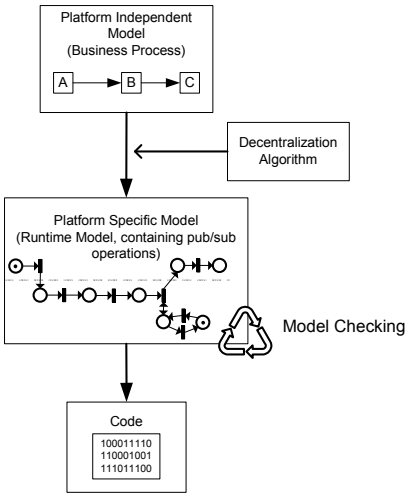


Figure 7: Model checking in MDA: from PIM to PSM

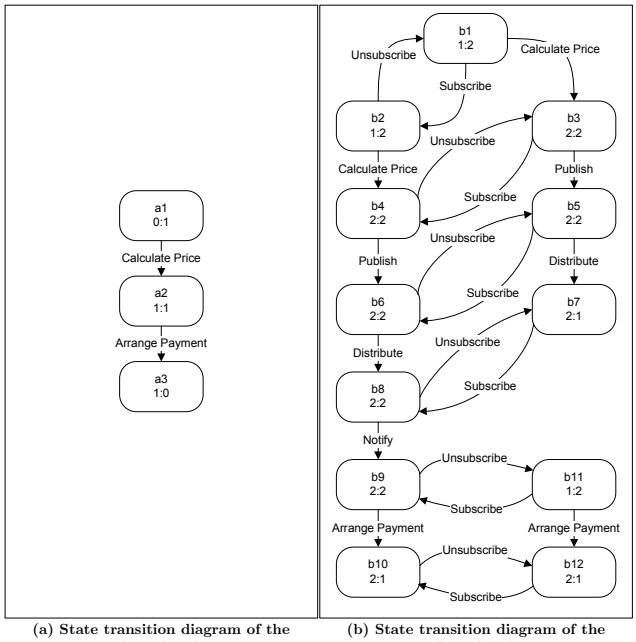
2. The petri net model also allows us to *model check* the event-based process execution, which includes the non-deterministic behavior of the publish/subscribe system itself (which would otherwise be left out). By including the publish/subscribe connection itself, we are sure we test the system like it would be eventually executed (using `publish`, `subscribe` and `notify` operations). This is illustrated in Figure 7, which shows the typical approach to Model Driven Architecture [16]. A platform independent model (the business process model) is transformed via a model to model transformation to a platform specific model, which again is transformed to code. The platform specific model represents the eventual execution of the process model. It is at this level that the publish/subscribe connection should be included and where model checking should be done. An example of model checking a publish/-subscribe process system is shown in Section 4.2.

Model checkers for petri-nets are widely available which can check the state space of a petri net for any required system properties. We can check for behavioral equivalences, user defined properties (defined in temporal logic), deadlocks, livelocks, race hazards, ... and we can also simulate the developed process flow, including the publish/subscribe connection, to calculate throughput, memory requirements or find bottlenecks.

4.1 Equivalence Checking Publish/Subscribe Process Execution: an Example

We give a small example of how the petri net *publish/-subscribe connection* can be used to check the equivalence of a designed process model and its running equivalent, which uses publish/subscribe communication between any distributed component. The eventual execution of the process model should always follow the original designed process model. Using a publish/subscribe connection in a petri net shouldn't structurally change the net itself.

Figure 6 shows two petri nets. Both nets indicate a sequence



(a) State transition diagram of the petri net in Figure

(b) State transition diagram of the petri net in Figure

Figure 8: State transition graph of a sequence between transitions *Calculate Price* and *Arrange Payment*, without and with the use of a publish-subscribe connection

of the two business tasks, *Calculate Price* and *Arrange Payment*. In figure 6a the process model is drawn as it would be designed by the process modeler, while in figure 6b, a publish-subscribe connection is used to accomplish the communication between the component running activity *Calculate Price* and the component running activity *Arrange Payment*. The state transition diagrams for these petri nets are shown in figure 8. We can now check the equivalence of these two models by using the following notion of branching bisimilarity [2]:

Definition 5. Two workflows W_1 and W_2 are branching bisimilar iff there exists a symmetric relation $\mathcal{R}$ such that: let τ be a silent step and $s \Rightarrow s'$ denote a path of zero or more τ steps from s to s':

1. The roots of the workflow are related by $\mathcal{R}$
2. $\mathcal{R}(r,s) \wedge r \xrightarrow{a} r' \Rightarrow \big(a = \tau \wedge \mathcal{R}(r',s)\big) \vee \big(\exists s : s \Rightarrow s_1 \xrightarrow{a} s' \wedge \mathcal{R}(r,s_1) \wedge \mathcal{R}(r',s')\big)$

With this notion we can see that the following bisimilarity relations hold:
$\mathcal{R}(a1,b1) \wedge \mathcal{R}(a1,b2) \wedge \mathcal{R}(a2,b2) \wedge \mathcal{R}(a2,b4) \wedge \mathcal{R}(a2,b5) \wedge \mathcal{R}(a2,b6) \wedge \mathcal{R}(a2,b7) \wedge \mathcal{R}(a2,b8) \wedge \mathcal{R}(a2,b9) \wedge \mathcal{R}(a2,b11) \wedge \mathcal{R}(a3,b10) \wedge \mathcal{R}(a3,b12)$

The two workflows (with and without the publish-subscribe connection) are thus observationally equivalent and have the same behavior. It is safe to deploy the process model and run it distributed with a publish/subscribe communication paradigm.

Table 3: Simulation of the pizza delivery process to measure the maximum memory size of the notification buffer for the *Bake Pizza* notifications

Time Interval of Bake Pizza	Time Interval of Deliver Pizza	Average notifications	Maximum Notifications	Maximum % of orders
5..15	1..10	2.7	4	0.8%
5..15	5..15	4	5	1%
5..15	10..20	4.6	6	12%
5..15	15..25	5.5	6	12%
5..15	20..30	6.9	7	14%
5..15	100..150	10.8	11	22%

4.2 Model Checking a Publish Subscribe Process System: an Example

We give an example of how a model simulation can be used to check certain properties of the resulting process execution, using publish/subscribe communication. Using simulation of the process model (including the pub/sub connections), we can figure out what the memory size of the buffer in the publish/subscribe connection should be for every event published. In this case, we like to calculate what the maximum buffer size of the *Bake Pizza* event notification should be for the *Deliver Pizza* activity (see figure 4). If the *Deliver Pizza* activity is slower than the *Bake Pizza* activity, event notifications have to be stored in the input buffer of the deliver pizza activity (place p_{2k} for connection $c : EC(c) =$ "*pizzabaked*"). The notifications are only consumed (and removed from the buffer place) when the activity *deliver pizza* is available again (e.g. the delivery boy returned from a trip).

We did this simulation using CPN/Tools [30], which is a toolset to model, analyze and simulate Colored Petri Nets. We developed the process model from Figure 4 in CPN/Tools and added a publish/subscribe connection between every component (so we achieve a platform specific model, see Figure 7). We ran a simulation for different time values for the activities *Bake Pizza* and *Deliver Pizza*. The net was simulated for 50 pizza orders that are received at a random time between 1 and 200 time units. 10 replications were done with each configuration, and per replication we measured the maximum amount of tokens (or event notifications) in the buffer place. Table 3 shows the results of the simulation. For this case we can see that, even if *pizza delivery* takes a substantial amount of time longer than *bake pizza*, the maximum amount of notifications stored in the buffer place for *deliver pizza* is 22% of the amount of orders over a certain time period.

Playing with the values of the net gives different simulations, and properties or problems can be checked beforehand. Not only simulations can be performed, but also other checks like reachability or checks for deadlocks can be examined (e.g. with the Model Checking Kit for petri nets [26]). These checks can be performed each time the net changes (e.g. publishers or subscribers are added or deleted from the system), in order to validate the change.

5. CONCLUSION

We proposed a petri net model that represents a publish/-subscribe communication between two or more components. This model can be included in a standard business process

model, so that a Platform Specific Model is obtained which symbolizes the runtime event-based process execution accurately. This model can then be used in model checking, where simulations can be performed, system properties can be retrieved and potential problems can be uncovered. We validated the petri net representation of a publish/subscribe communication by checking its correctness with a number of publish/subscribe properties (e.g. its safety and liveness conditions). We also showed two examples of how the model, incorporated in a business process model, can be used to model check the event-based process execution: we showed how the behavioral equivalence between the original model and the runtime model can be checked and showed how a simulation can be used to retrieve a memory requirement property of the publish/subscribe service.

6. REFERENCES

[1] R. Baldoni, M. Contenti, S. Piergiovanni, and A. Virgillito. Modeling publish/subscribe communication systems: towards a formal approach. In *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems, 2003.(WORDS 2003)*, pages 304–311, 2003.

[2] T. Basten. Branching Bisimilarity is an Equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.

[3] G. Chafle, S. Chandra, V. Mann, and M. Nanda. Decentralized orchestration of composite web services. *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 134–143, 2004.

[4] E. Clarke. Model checking. *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56, 1997.

[5] W. M. Coalition. The workflow reference model. *WfMC Documents*, 1995.

[6] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet computing*, 6(2):86–93, 2002.

[7] X. Deng, M. Dwyer, J. Hatcliff, G. Jung, and G. Singh. Model-checking middleware-based event-driven real-time embedded software. pages 154–181, 2003.

[8] M. Dumas, W. Van Der Aalst, and A. Ter Hofstede. *Process-aware information systems*. Wiley-Interscience, 2005.

[9] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):131, 2003.

[10] D. Garlan, S. Khersonsky, and J. Kim. Model checking publish-subscribe systems. *Model Checking Software*, pages 623–623, 2003.

[11] P. Hens, M. Snoeck, M. De Backer, and G. Poels. Decentralized Event-Based Orchestration. In *Inter. Work. on Event-Driven Business Process Management*, 2010.

[12] P. Hens, M. Snoeck, M. De Backer, and G. Poels. An Event-Based Architecture for Distributed Process Execution: feasability and flexibility assesment. In *submitted for the 1st conference on Business Process Modeling, Development and Support (BPMDS)*, 2011.

[13] O. Ibe and K. Trivedi. Stochastic petri net models of polling systems. *Selected Areas in Communications, IEEE Journal on*, 8(9):1649 –1657, Dec. 1990.

[14] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods, and practical use*. Springer, 1996.

[15] P. Katsaros, V. Odontidis, and M. Gousidou-Koutita. Colored Petri Net based model checking and failure analysis for E-commerce protocols. page 267, 2005.

[16] A. Kleppe, J. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[17] Q. Li, H. Zhu, J. Li, and J. He. Scalable formalization of publish/subscribe messaging scheme based on message brokers. *Web Services and Formal Methods*, pages 61–76, 2008.

[18] Z. Li and M. Xiaodong. A kind of Petri net model for analyzing C3I communication system. 2:237–239.

[19] B. Michelson. Event-driven architecture overview. *OMG report*, 2006.

[20] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2006.

[21] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, 2002.

[22] P. Muth, D. Wodtke, J. Weissenfels, A. Dittrich, and G. Weikum. From centralized workflow specification to distributed workflow execution. *Journal of Intelligent Information Systems*, 10(2):159–184, 1998.

[23] Oasis. Web service business process execution language version 2.0. Oasis Standard.

[24] Object Management Group. Bpmn 2.0, beta 2. http://www.omg.org/cgi-bin/doc?dtc/10-06-04, June 2010.

[25] K. Salimifard and M. Wright. Petri net-based modelling of workflow systems: An overview. *European journal of operational research*, 134(3):664–676, 2001.

[26] C. Schröter, S. Schwoon, and J. Esparza. The model-checking kit. In *Proceedings of the 24th international conference on Applications and theory of Petri nets*, ICATPN'03, pages 463–472, Berlin, Heidelberg, 2003. Springer-Verlag.

[27] B. Sriraman, L. Architect, R. Radhakrishnan, and E. Architect. Event Driven Architecture Augmenting Service Oriented Architectures. *Sun Microsystems*, 2005.

[28] W. Van der Aalst et al. The application of Petri nets to workflow management. *Journal of Circuits Systems and Computers*, 8:21–66, 1998.

[29] W. Van Der Aalst and A. Ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[30] A. Vinter Ratzer, L. Wells, H. Lassen, M. Laursen, J. Qvortrup, M. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN tools for editing, simulating, and analysing coloured Petri nets. pages 1024–1024, 2003.

[31] L. Zanolin, C. Ghezzi, and L. Baresi. An approach to model and validate publish/subscribe architectures. *SAVCBS 2003 Specification and Verification of Component-Based Systems*, page 35, 2003.